Hacking the Selenium WebDriver

A deep dive into WebDriver's Python implementation

Daniel Puterman (LinkedIn), Director of R&D, Applitools

(first ever!) PyCon Israel 2016

On the agenda

- An overview of Selenium/Webdriver & the Wire Protocol
- Hacking the Python implementation: write an SDK which records the user actions for later playback.

Selenium Overview
(but first, the video demo)

Selenium automates browsers

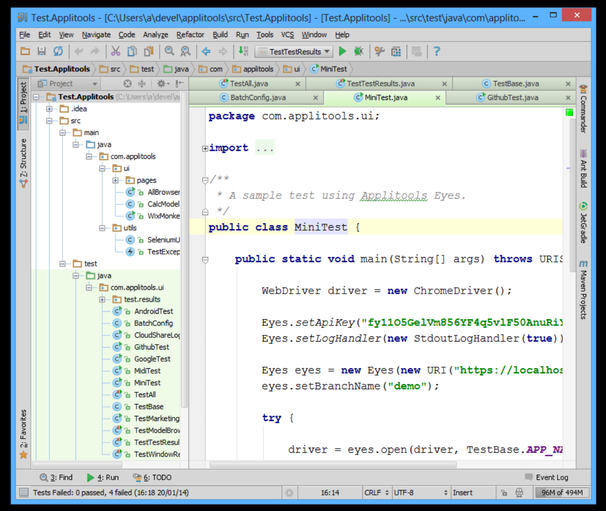
On all major OS

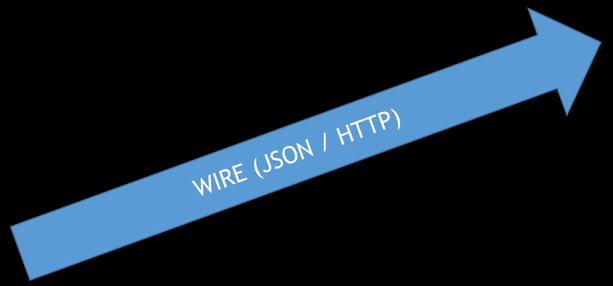
And mobile… (e.g. via Appium)

Open source under the Apache License.

No proprietary IDE/programming language

How does it work





webdriver.xpi

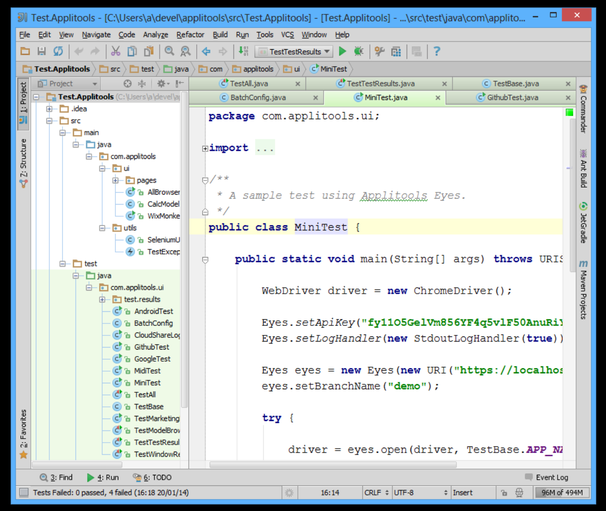
XPCOM

IEDriverServer.exe

COM

How does it work

What we care about



WIRE (JSON / HTTP)

WIRE (JSON / HTTP)

webdriver.xpi

XPCOM

IEDriverServer.exe

COM

What's Wire?

A RESTFul web service using JSON over HTTP.
A proposed W3C standard for automating web-browsers.

https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

/session/:sessionId/screenshot

GET /session/:sessionId/screenshot

Take a screenshot of the current page.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{string}` The screenshot as a base64 encoded PNG.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

On the agenda

- An overview of Selenium/Webdriver & the Wire Protocol
- Hacking the Python implementation: write an SDK which records the user actions for later playback.

The Benchmark Script

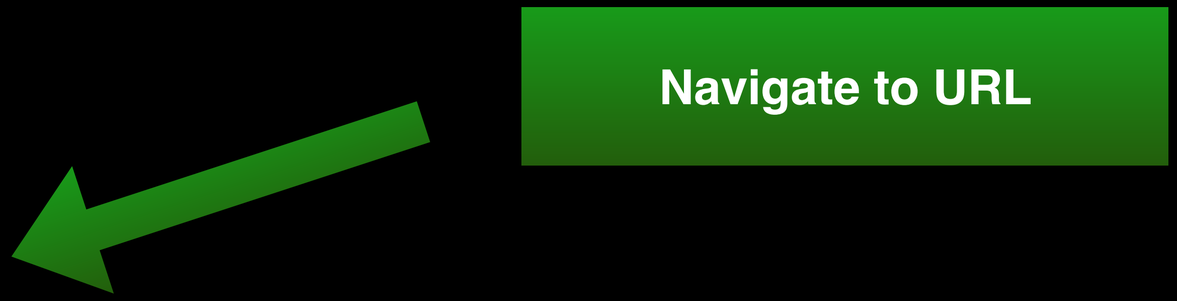
```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('sign up to our list’)

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

Functional overview

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('sign up to our list’)

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

Open up the browser

Functional overview




```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('sign up to our list’)

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

Functional overview

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('sign up to our list’)

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

Find the “waiting list” page link

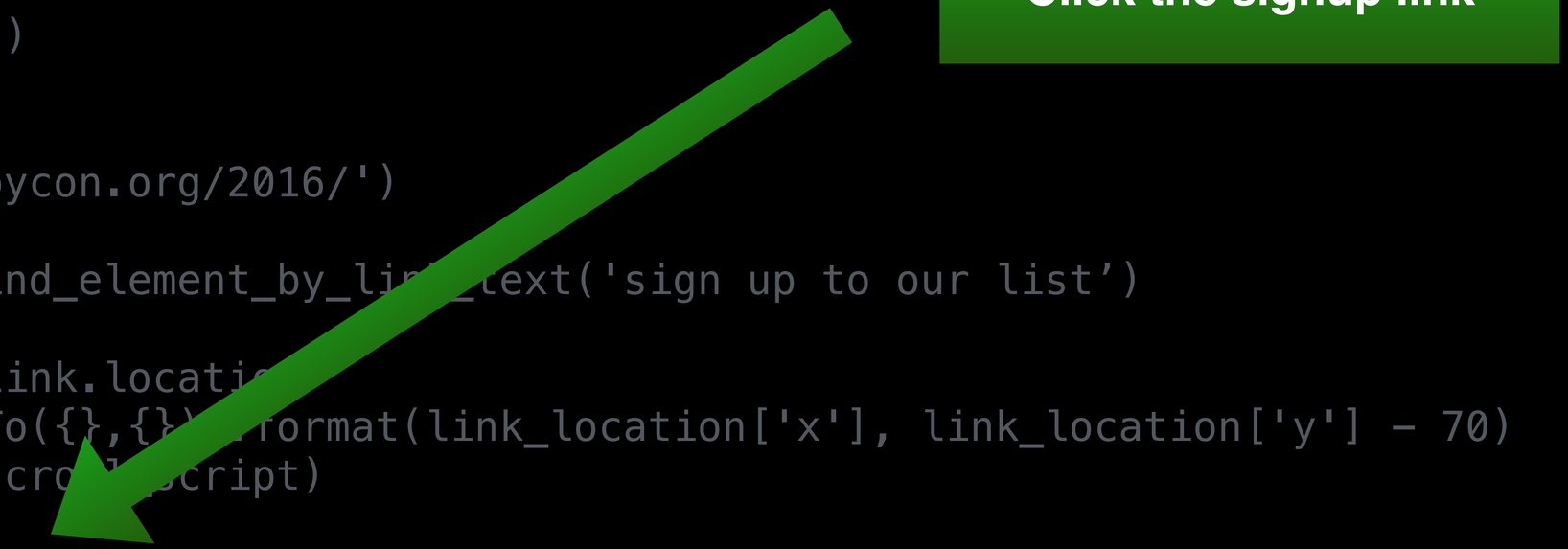
Functional overview

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('sign up to our list')

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

Scroll to the link location (almost)

Functional overview

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('sign up to our list')

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```



Functional overview

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('Sign up to our list')

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

Find and click the “join the waiting list” button

Functional overview

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('Sign up to our list')

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

Type “Selenium Webdriver” in the “name” input

Functional overview

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_te('sign up to our list')

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".for(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_scri

    signup_link.click()

    browser.find_element_by_ss_name('btn-waitlist').click()

    browser.find_eleme by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

Close the browser

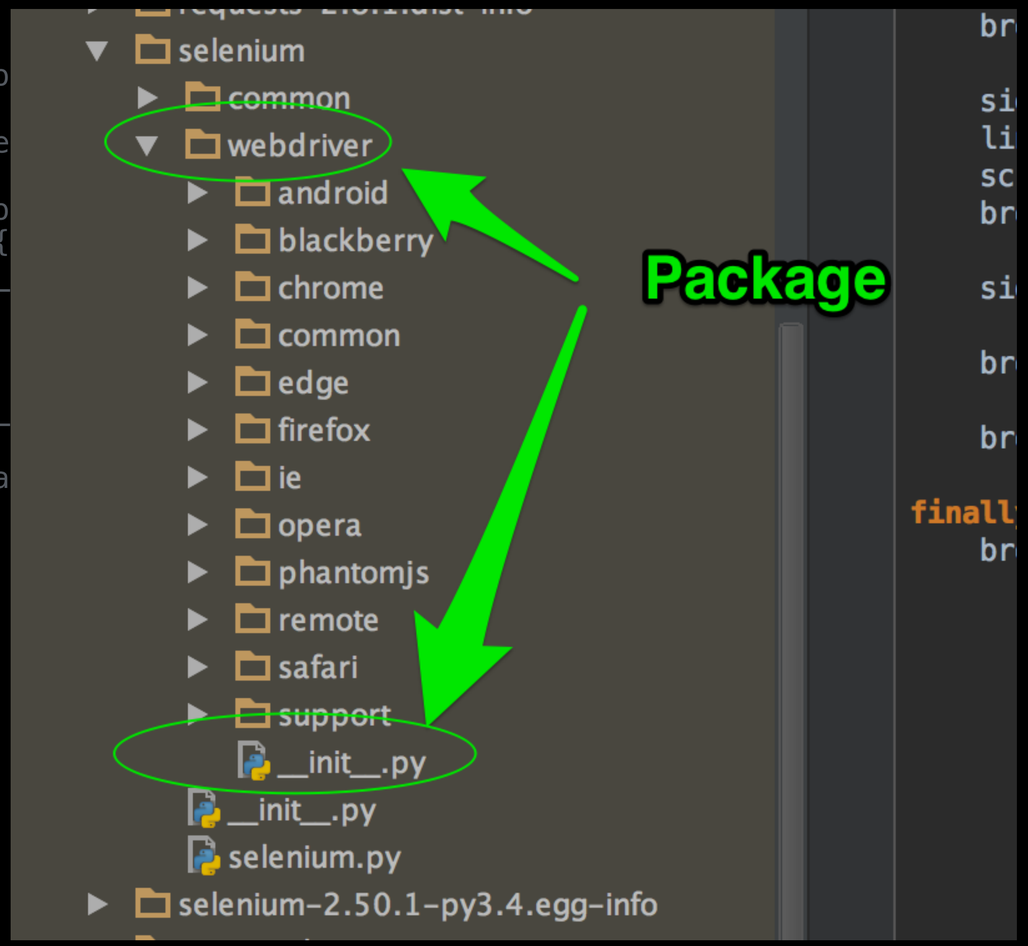
Our lightweight SDK

- Track WebElement actions: click, send_keys
- Track navigation (i.e., associate actions with page): get.
- Export actions in JSON format.

Down the rabbit hole we go…

webdriver

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.o

    signup_link = browser.find_ele

    link_location = signup_link.lo
    scroll_script = "scrollTo({},{
    browser.execute_script(scroll_

    signup_link.click()

    browser.find_element_by_class_

    browser.find_element_by_id('wa

finally:
    browser.quit()
```



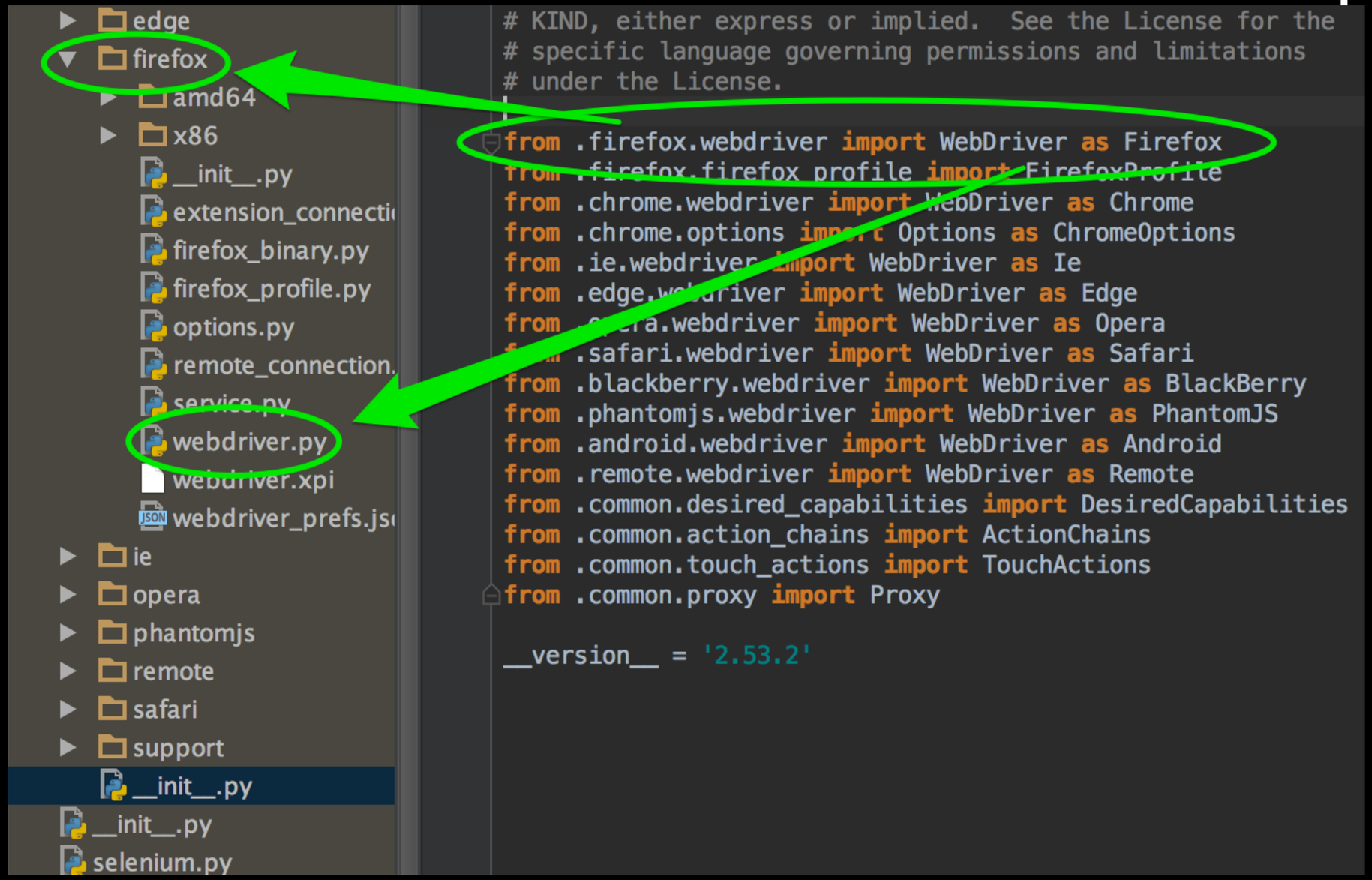
Class? Module? Package?

Opening up the browser

```
from selenium import webdriver

browser = webdriver.Firefox()

try:
    browser.get('http://il.pycon.org/2016/')

    signup_link = browser.find_element_by_link_text('sign up to our list')

    link_location = signup_link.location
    scroll_script = "scrollTo({},{})".format(link_location['x'], link_location['y'] - 70)
    browser.execute_script(scroll_script)

    signup_link.click()

    browser.find_element_by_class_name('btn-waitlist').click()

    browser.find_element_by_id('waitlisted_person_name').send_keys('Selenium WebDriver')

finally:
    browser.quit()
```

So what's this?

selenium/webdriver/__init__.py



selenium/webdriver/firefox/webdrive.py

```
# Python 2/3 compat + imports…
from selenium.webdriver.remote.webdriver import WebDriver as RemoteWebDriver

class WebDriver(RemoteWebDriver):

    # There is no native event support on Mac
    NATIVE_EVENTS_ALLOWED = sys.platform != "darwin"

    def __init__(self, firefox_profile=None, firefox_binary=None, timeout=30,
                 capabilities=None, proxy=None, executable_path="wires", firefox_options=None):
      # ...
            executor = ExtensionConnection("127.0.0.1", self.profile, self.binary, timeout)
            RemoteWebDriver.__init__(self,
                command_executor=executor,
            desired_capabilities=capabilities,
            keep_alive=True)

        self._is_remote = False

    def quit(self):
        """Quits the driver and close every associated window."""
   # ...

    @property
    def firefox_profile(self):
        return self.profile

    def set_context(self, context):
        self.execute("SET_CONTEXT", {"context": context})
```

selenium/webdriver/firefox/webdrive.py

```
# Python 2/3 compat + imports…
from selenium.webdriver.remote.webdriver import WebDriver as RemoteWebDriver

class WebDriver(RemoteWebDriver):

    # There is no native event support on Mac
    NATIVE_EVENTS_ALLOWED = sys.platform != "darwin"

    def __init__(self, firefox_profile=None, firefox_binary=None, timeout=30,
                 capabilities=None, proxy=None, executable_path="wires", firefox_options=None):
      # ...
            executor = ExtensionConnection("127.0.0.1", self.profile, self.binary, timeout)
            RemoteWebDriver.__init__(self,
                command_executor=executor,
            desired_capabilities=capabilities,
            keep_alive=True)

        self._is_remote = False

    def quit(self):
        """Quits the driver and close every associated window."""
    # ...

    @property
    def firefox_profile(self):
        return self.profile

    def set_context(self, context):
        self.execute("SET_CONTEXT", {"context": context})
```

Basically a thin wrapper around “RemoteWebDriver”

selenium/webdriver/__init__.py

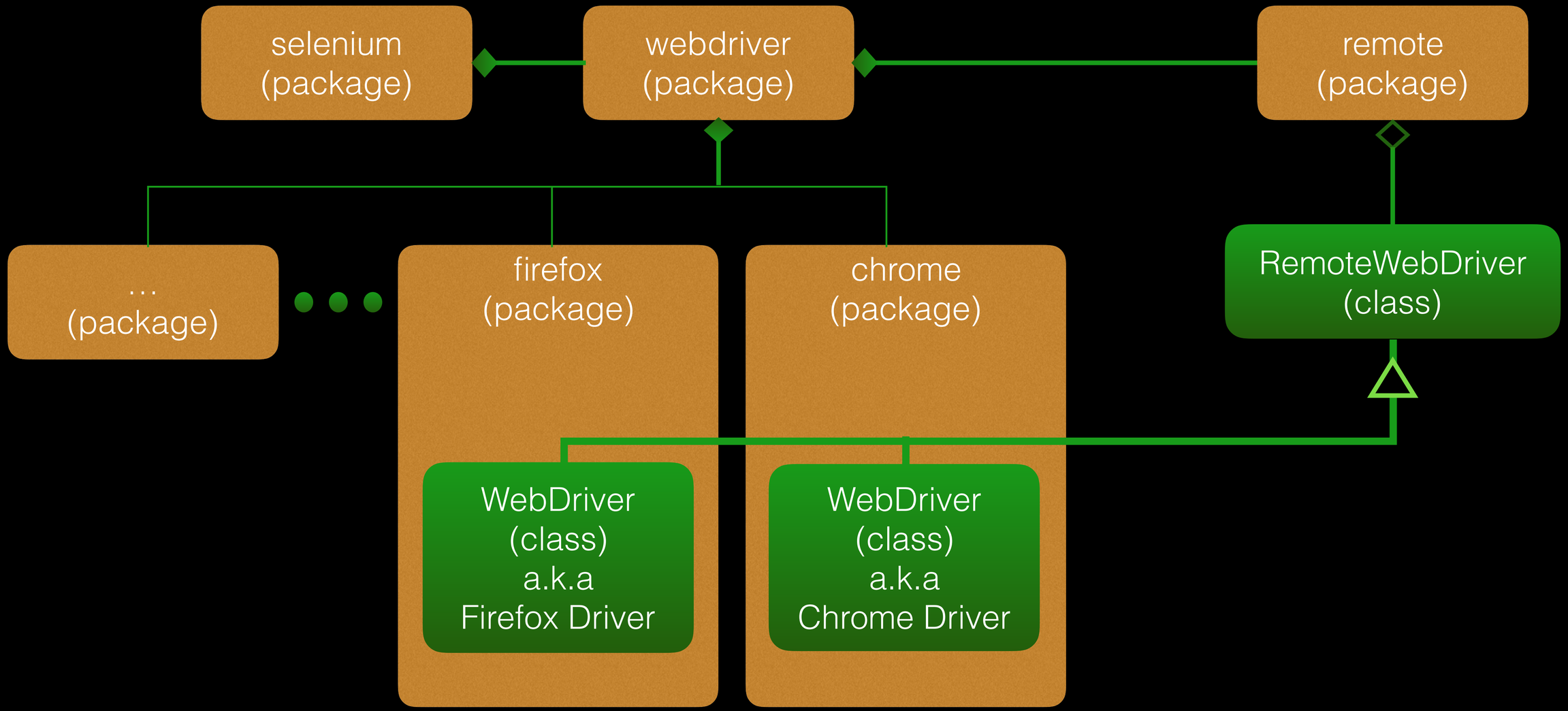
```
from .firefox.webdriver import WebDriver as Firefox
from .firefox.firefox_profile import FirefoxProfile
from .chrome.webdriver import WebDriver as Chrome
from .chrome.options import Options as ChromeOptions
from .ie.webdriver import WebDriver as Ie
from .edge.webdriver import WebDriver as Edge
from .opera.webdriver import WebDriver as Opera
from .safari.webdriver import WebDriver as Safari
from .blackberry.webdriver import WebDriver as BlackBerry
from .phantomjs.webdriver import WebDriver as PhantomJS
from .android.webdriver import WebDriver as Android
from .remote.webdriver import WebDriver as Remote
from .common.desired_capabilities import DesiredCapabilities
from .common.action_chains import ActionChains
from .common.touch_actions import TouchActions
from .common.proxy import Proxy
```

Other types of drivers are (mostly) thin wrappers as well

selenium/webdriver/__init__.py

```
from .firefox.webdriver import WebDriver as Firefox
from .firefox.firefox_profile import FirefoxProfile
from .chrome.webdriver import WebDriver as Chrome
from .chrome.options import Options as ChromeOptions
from .ie.webdriver import WebDriver as Ie
from .edge.webdriver import WebDriver as Edge
from .opera.webdriver import WebDriver as Opera
from .safari.webdriver import WebDriver as Safari
from .blackberry.webdriver import WebDriver as BlackBerry
from .phantomjs.webdriver import WebDriver as PhantomJS
from .android.webdriver import WebDriver as Android
from .remote.webdriver import WebDriver as Remote
from .common.desired_capabilities import DesiredCapabilities
from .common.action_chains import ActionChains
from .common.touch_actions import TouchActions
from .common.proxy import Proxy
```

So what do we have so far?



selenium/webdriver/remote/webdriver.py
(a.k.a RemoteWebDriver)

selenium/webdriver/remote/webdriver.py

```
"""The WebDriver implementation."""
# imports ...
class WebDriver(object):
    """
    Controls a browser by sending commands to a remote server.
    This server is expected to be running the WebDriver wire protocol
    as defined at
    https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

    :Attributes:
     - session_id - String ID of the browser session started and controlled by this WebDriver.
     - capabilities - Dictionaty of effective capabilities of this browser session as returned
         by the remote server.
         See https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities
     - command_executor - remote_connection.RemoteConnection object used to execute commands.
     - error_handler - errorhandler.ErrorHandler object used to handle errors.
    """

    def __init__(self, command_executor='http://127.0.0.1:4444/wd/hub',
                 desired_capabilities=None, browser_profile=None, proxy=None, keep_alive=False,
                 file_detector=None):
        ...
```

selenium/webdriver/remote/webdriver.py

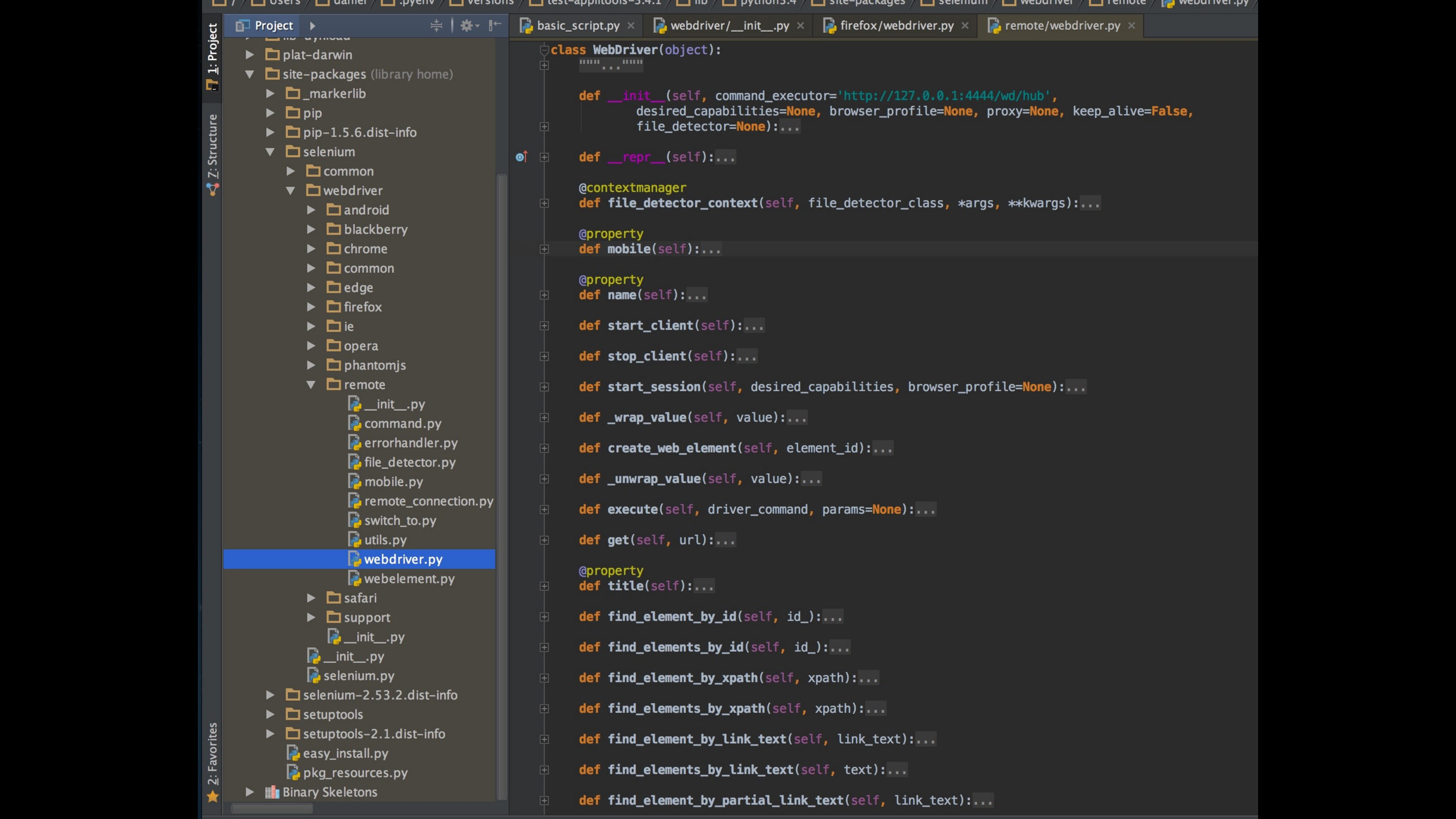
Yay!

```
"""The WebDriver implementation."""
# imports ...
class WebDriver(object):
    """
    Controls a browser by sending commands to a remote server.
    This server is expected to be running the WebDriver wire protocol
    as defined at
    https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

    :Attributes:
     - session_id - String ID of the browser session started and controlled by this WebDriver.
     - capabilities - Dictionaty of effective capabilities of this browser session as returned
         by the remote server.
         See https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities
     - command_executor - remote_connection.RemoteConnection object used to execute commands.
     - error_handler - errorhandler.ErrorHandler object used to handle errors.
    """

    def __init__(self, command_executor='http://127.0.0.1:4444/wd/hub',
            desired_capabilities=None, browser_profile=None, proxy=None, keep_alive=False,
            file_detector=None):
        ...
```

selenium/webdriver/remote/webdriver.py

So, not complicated

```
"""The WebDriver implementation."""
# imports ...
class WebDriver(object):
    """
    Controls a browser by sending commands to a remote server.
    This server is expected to be running the WebDriver wire protocol
    as defined at
    https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

    :Attributes:
     - session_id - String ID of the browser session started and controlled by this WebDriver.
     - capabilities - Dictionaty of effective capabilities of this browser session as returned
         by the remote server.
         See https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities
     - command_executor - remote_connection.RemoteConnection object used to execute commands.
     - error_handler - errorhandler.ErrorHandler object used to handle errors.
    """

    def __init__(self, command_executor='http://127.0.0.1:4444/wd/hub',
                 desired_capabilities=None, browser_profile=None, proxy=None, keep_alive=False,
                 file_detector=None):
        ...
```


Project
basic_script.py
webdriver/__init__.py
firefox/webdriver.py
remote/webdriver.py
plat-darwin
site-packages (library home)
_markerlib
pip
pip-1.5.6.dist-info
selenium
common
webdriver
android
blackberry
chrome
common
edge
firefox
ie
opera
phantomjs
remote
__init__.py
command.py
errorhandler.py
file_detector.py
mobile.py
remote_connection.py
switch_to.py
utils.py
webdriver.py
webelement.py
safari
support
__init__.py
__init__.py
selenium.py
selenium-2.53.2.dist-info
setuptools
setuptools-2.1.dist-info
easy_install.py
pkg_resources.py
Binary Skeletons
class WebDriver(object):
"""..."""
def __init__(self, command_executor='http://127.0.0.1:4444/wd/hub',
desired_capabilities=None, browser_profile=None, proxy=None, keep_alive=False,
file_detector=None):...
def __repr__(self):...
@contextmanager
def file_detector_context(self, file_detector_class, *args, **kwargs):...
@property
def mobile(self):...
@property
def name(self):...
def start_client(self):...
def stop_client(self):...
def start_session(self, desired_capabilities, browser_profile=None):...
def _wrap_value(self, value):...
def create_web_element(self, element_id):...
def _unwrap_value(self, value):...
def execute(self, driver_command, params=None):...
def get(self, url):...
@property
def title(self):...
def find_element_by_id(self, id_):...
def find_elements_by_id(self, id_):...
def find_element_by_xpath(self, xpath):...
def find_elements_by_xpath(self, xpath):...
def find_element_by_link_text(self, link_text):...
def find_elements_by_link_text(self, text):...
def find_element_by_partial_link_text(self, link_text):...

selenium/webdriver/remote/webdriver.py

- Initialization code
- A bunch of commands (get/set cookies, get screenshot etc.).
- Navigation methods
- A lot of “find_elementXXXX” variants
- A bunch of properties

selenium/webdriver/remote/webdriver.py

What we're interested in

- Initialization code
- A bunch of commands (get/set cookies, get screenshot etc.)
- Navigation methods
- A lot of "find_elementXXXX" variants
- A bunch of properties

selenium/webdriver/remote/webdriver.py

- Initialization code
- A bunch of commands (get/set cookies, get screenshot etc.)
- Navigation methods
- A lot of “find_elementXXXX” variants
- A bunch of properties

RemoteWebDriver: navigation methods

```
def get(self, url):
    """…"""
    self.execute(Command.GET, {'url': url})

...

#Navigation
def back(self):
    """…"""
    self.execute(Command.GO_BACK)

def forward(self):
    """…"""
    self.execute(Command.GO_FORWARD)

def refresh(self):
    """…"""
    self.execute(Command.REFRESH)
```

RemoteWebDriver: navigation methods

```
def get(self, url):
    """…"""
    self.execute(Command.GET, {'url': url})

...

#Navigation
def back(self):
    """…"""
    self.execute(Command.GO_BACK)

def forward(self):
    """…"""
    self.execute(Command.GO_FORWARD)

def refresh(self):
    """…"""
    self.execute(Command.REFRESH)
```

So method calls are wrappers for ‘self.execute’

no complex logic is need for our SDK wrapper

‘RemoteWebDriver’: a quick look at ‘self.execute’

```
def execute(self, driver_command, params=None):
    """Sends a command to be executed by a
            command.CommandExecutor.
    # ...
    :Returns:
      The command's JSON response loaded into a dictionary object.
    """
    if self.session_id is not None:
        if not params:
            params = {'sessionId': self.session_id}
        elif 'sessionId' not in params:
            params['sessionId'] = self.session_id

    params = self._wrap_value(params)
    response = self.command_executor.execute(driver_command, params)
    if response:
        self.error_handler.check_response(response)
        response['value'] = self._unwrap_value(
            response.get('value', None))
        return response
    # If the server doesn't send a response, assume the command was
    # a success
    return {'success': 0, 'value': None, 'sessionId': self.session_id}
```

As expected…

‘RemoteWebDriver’: a quick look at ‘self.execute’

```
def execute(self, driver_command, params=None):
    """Sends a command to be executed by a
            command.CommandExecutor.
    # ...
    :Returns:
      The command's JSON response loaded into a dictionary obje
    """
    if self.session_id is not None:
        if not params:
            params = {'sessionId': self.session_id}
        elif 'sessionId' not in params:
            params['sessionId'] = self.session_id

    params = self._wrap_value(params)
    response = self.command_executor.execute(driver_command, params)
    if response:
        self.error_handler.check_response(response)
        response['value'] = self._unwrap_value(
            response.get('value', None))
        return response
    # If the server doesn't send a response, assume the command was
    # a success
    return {'success': 0, 'value': None, 'sessionId': self.session_id}
```

As expected…

Grab the session ID

‘RemoteWebDriver’: a quick look at ‘self.execute’

```
def execute(self, driver_command, params=None):
    """Sends a command to be executed by a
            command.CommandExecutor.
    # ...
    :Returns:
      The command's JSON response loaded into a dictionary obje
    """
    if self.session_id is not None:
        if not params:
            params = {'sessionId': self.session_id}
        elif 'sessionId' not in params:
            params['sessionId'] = self.session_id

    params = self._wrap_value(params)
    response = self.command_executor.execute(driver_command, params)
    if response:
        self.error_handler.check_response(response)
        response['value'] = self._unwrap_value(
            response.get('value', None))
        return response
    # If the server doesn't send a response, assume the command was
    # a success
    return {'success': 0, 'value': None, 'sessionId': self.session_id}
```

As expected…

Grab the session ID

Send the command to the driver server

‘RemoteWebDriver’: a quick look at ‘self.execute’

```
def execute(self, driver_command, params=None):
    """Sends a command to be executed by a
            command.CommandExecutor.
    # ...
    :Returns:
      The command's JSON response loaded into a dictionary obje
    """
    if self.session_id is not None:
        if not params:
            params = {'sessionId': self.session_id}
        elif 'sessionId' not in params:
            params['sessionId'] = self.session_id

    params = self._wrap_value(params)
    response = self.command_executor.execute(drive      mand, params)
    if response:
        self.error_handler.check_response(response)
        response['value'] = self._unwrap_value(
            response.get('value', None))
        return response
    # If the server doesn't send a response, assume the command was
    # a success
    return {'success': 0, 'value': None, 'sessionId': self.session_id}
```

As expected…

Grab the session ID

Send the command to the driver server

Return the response

selenium/webdriver/remote/webdriver.py

- Initialization code
- A bunch of commands (get/set cookies, get screenshot etc.)
- Navigation methods
- A lot of “find_elementXXXX” variants
- A bunch of properties

RemoteWebDriver: ‘findElementXXX’ methods

```
# ...

def find_element_by_id(self, id_):
    return self.find_element(by=By.ID, value=id_)

def find_elements_by_id(self, id_):
    return self.find_elements(by=By.ID, value=id_)

def find_element_by_xpath(self, xpath):
   return self.find_element(by=By.XPATH, value=xpath)

def find_elements_by_xpath(self, xpath):
    return self.find_elements(by=By.XPATH, value=xpath)

def find_element_by_link_text(self, link_text):
    return self.find_element(by=By.LINK_TEXT, value=link_text)

# ...
```

Basically wrappers for 2 types of methods

RemoteWebDriver: ‘findElementXXX’ methods

```
# ...

def find_element_by_id(self, id_):
    return self.find_element(by=By.ID, value=id_)

def find_elements_by_id(self, id_):
    return self.find_elements(by=By.ID, value=id_)

def find_element_by_xpath(self, xpath):
   return self.find_element(by=By.XPATH, value=xpath)

def find_elements_by_xpath(self, xpath):
    return self.find_elements(by=By.XPATH, value=xpath)

def find_element_by_link_text(self, link_text):
    return self.find_element(by=By.LINK_TEXT, value=link_text)

# ...
```

Basically wrappers for 2 types of methods

Find a single element

RemoteWebDriver: ‘findElementXXX’ methods

```
# ...

def find_element_by_id(self, id_):
    return self.find_element(by=By.ID, value=id_)

def find_elements_by_id(self, id_):
    return self.find_elements(by=By.ID, value=id_)

def find_element_by_xpath(self, xpath):
   return self.find_element(by=By.XPATH, value=xpath)

def find_elements_by_xpath(self, xpath):
    return self.find_elements(by=By.XPATH, value=xpath)

def find_element_by_link_text(self, link_text):
    return self.find_element(by=By.LINK_TEXT, value=link_text)

# ...
```

Basically wrappers for 2 types of methods

Find a single element

Find multiple elements

RemoteWebDriver: ‘self.find_element’

```
def find_element(self, by=By.ID, value=None):
    """
    # ...

    :rtype: WebElement
    """
    if not By.is_valid(by) or not isinstance(value, str):
        raise InvalidSelectorException("Invalid locator values passed in")
    if self.w3c:
        if by == By.ID:
            by = By.CSS_SELECTOR
            value = '[id="%s"]' % value
        elif by == By.TAG_NAME:
            by = By.CSS_SELECTOR
        elif by == By.CLASS_NAME:
            by = By.CSS_SELECTOR
            value = ".%s" % value
        elif by == By.NAME:
            by = By.CSS_SELECTOR
            value = '[name="%s"]' % value
    return self.execute(Command.FIND_ELEMENT,
                        {'using': by, 'value': value})['value']
```

RemoteWebDriver: ‘self.find_element’

```
def find_element(self, by=By.ID, value=None):
    """
    # ...

    :rtype: WebElement
    """
    if not By.is_valid(by) or not isinstance(value, str):
        raise InvalidSelectorException("Invalid locator values passed in")
    if self.w3c:
        if by == By.ID:
            by = By.CSS_SELECTOR
            value = '[id="%s"]' % value
        elif by == By.TAG_NAME:
            by = By.CSS_SELECTOR
        elif by == By.CLASS_NAME:
            by = By.CSS_SELECTOR
            value = ".%s" % value
        elif by == By.NAME:
            by = By.CSS_SELECTOR
            value = '[name="%s"]' % value
    return self.execute(Command.FIND_ELEMENT,
                        {'using': by, 'value': value})['value']
```

We’ll want to wrap this with our own WebElement

RemoteWebDriver: ‘self.find_element**s**’

```
def find_elements(self, by=By.ID, value=None):
    """
    # ...
   :rtype: list of WebElement
    """
    if not By.is_valid(by) or not isinstance(value, str):
        raise InvalidSelectorException("Invalid locator values passed in")
    if self.w3c:
        if by == By.ID:
            by = By.CSS_SELECTOR
            value = '[id="%s"]' % value
        elif by == By.TAG_NAME:
            by = By.CSS_SELECTOR
        elif by == By.CLASS_NAME:
            by = By.CSS_SELECTOR
            value = ".%s" % value
        elif by == By.NAME:
            by = By.CSS_SELECTOR
            value = '[name="%s"]' % value

    return self.execute(Command.FIND_ELEMENTS,
                        {'using': by, 'value': value})['value']
```

RemoteWebDriver: ‘self.find_element**s**’

```
def find_elements(self, by=By.ID, value=None):
    """
    # ...
   :rtype: list of WebElement
    """
    if not By.is_valid(by) or not isinstance(value, str):
        raise InvalidSelectorException("Invalid locator values passed in")
    if self.w3c:
        if by == By.ID:
            by = By.CSS_SELECTOR
            value = '[id="%s"]' % value
        elif by == By.TAG_NAME:
            by = By.CSS_SELECTOR
        elif by == By.CLASS_NAME:
            by = By.CSS_SELECTOR
            value = ".%s" % value
        elif by == By.NAME:
            by = By.CSS_SELECTOR
            value = '[name="%s"]' % value

    return self.execute(Command.FIND_ELEMENTS,
                        {'using': by, 'value': value})['value']
```

We’ll want to wrap <u>each element in the list</u>

selenium/webdriver/remote/webelement.py

(a short break from RemoteWebDriver)

selenium/webdriver/remote/webelement.py

```
class WebElement(object):
...
    def click(self):
     """Clicks the element."""
     self._execute(Command.CLICK_ELEMENT)
    ...
    def find_element(self, by=By.ID, value=None):
        #...
        return self._execute(Command.FIND_CHILD_ELEMENT,{"using": by, "value": value})['value']

    def find_elements(self, by=By.ID, value=None):
        #...
        return self._execute(Command.FIND_CHILD_ELEMENTS,{"using": by, "value": value}['value']

    @property
    def location(self):
        # ...

    @property
    def rect(self):
        # ...
        else:
            return self._execute(Command.GET_ELEMENT_RECT)['value']
```

selenium/webdriver/remote/webdriver.py

Very similar to ‘RemoteWebDriver’

- Initialization code
- A bunch of commands (click, send_keys etc.)
- A lot of “find_elementXXXX” variants
- A bunch of properties

selenium/webdriver/remote/webdriver.py

What we're interested in

- Initialization code
- A bunch of commands (click, send_keys etc.)
- A lot of "find_elementXXXX" variants
- A bunch of properties

selenium/webdriver/remote/webdriver.py

- Initialization code
- A bunch of commands (click, send_keys etc.)
- A lot of “find_elementXXXX” variants
- A bunch of properties

Requires special handling

https://mail.python.org/pipermail/python-dev/2005-December/058498.html

On 12/7/05, Greg Ewing <greg.ewing at canterbury.ac.nz> wrote:

```
> Maybe descriptors need a fourth slot for hasattr
> customisation?
>
> The logic would then be
>
>    if there is a descriptor for the attribute:
>      if the descriptor's hasattr slot is populated:
>        return the result of calling it
>      else:
>        return True
>    else:
>      look in the instance dict for the attribute
```

Um, that does't work for types which customize __getattribute__ or __getattr__ in various ways.

IMO a property that has a side effect (other than updating a cache or statistics or perhaps logging) is a misfeature anyway, so I don't see what's wrong with hasattr() trying getattr() and reporting False IFF that raises an exception.

If you want only AttributeError to be handled, use getattr(x, 'name', None).

--
--Guido van Rossum (home page: http://www.python.org/~guido/)

selenium/webdriver/remote/webelement.py

:-(

```
class WebElement(object):
...
    def click(self):
     """Clicks the element."""
     self._execute(Command.CLICK_ELEMENT)
    ...
    def find_element(self, by=By.ID, value=None):
        #...
        return self._execute(Command.FIND_CHILD_ELEMENT,{"using": by, "value": value})['value']

    def find_elements(self, by=By.ID, value=None):
        #...
        return self._execute(Command.FIND_CHILD_ELEMENTS,{"using": by, "value": value}['value']

    @property
    def location(self):
        # ...

    @property
    def rect(self):
        # ...
        else:
            return self._execute(Command.GET_ELEMENT_RECT)['value']
```

Got enough info! Let’s code!

meta_prog.py

meta_prog_utils.py

```
def create_proxy_interface(from_, to, ignore_list=None, override_existing=False):
    """
    Copies the public interface of the destination object, excluding names in the ignore_list,
    and creates an identical interface in 'src', which forwards calls to dst.
    If 'override_existing' is False, then attributes already existing in 'src' will not be
    overridden.
    """
    if not ignore_list:
        ignore_list = []
    for attr_name in dir(to):
        if not attr_name.startswith('_') and not attr_name in ignore_list:
            if callable(getattr(to, attr_name)):
                if override_existing or not hasattr(from_, attr_name):
                    setattr(from_, attr_name, create_forwarded_method(from_, to, attr_name))
```

meta_prog_utils.py

```
def create_proxy_interface(from_, to, ignore_list=None, override_exi
    """
    Copies the public interface of the destination object, excluding
    and creates an identical interface in 'src', which forwards calls to dst.
    If 'override_existing' is False, then attributes already existing in 'src' will not be
    overridden.
    """
    if not ignore_list:
        ignore_list = []
    for attr_name in dir(to):
        if not attr_name.startswith('_') and not attr_name in ignore_list:
            if callable(getattr(to, attr_name)):
                if override_existing or not hasattr(from_, attr_name):
                    setattr(from_, attr_name, create_forwarded_method(from_, to, attr_name))
```

dirty handling properties. cuz, h4x0r

meta_prog_utils.py

```
def create_proxy_interface(from_, to, ignore_list=None, override_existing=False):
    """
    Copies the public interface of the destination object, excluding names in the ignore_list,
    and creates an identical interface in 'src', which forwards calls to dst.
    If 'override_existing' is False, then attributes already existing in 'src' will not be
    overridden.
    """
    if not ignore_list:
        ignore_list = []
    for attr_name in dir(to):
        if not attr_name.startswith('_') and not attr_name in ignore_list:
            if callable(getattr(to, attr_name)):
                if override_existing or not hasattr(from_, attr_name):
                    setattr(from_, attr_name, create_forwarded_method(from_, to, attr_name))
```

meta_prog_utils.py

```
def create_forwarded_method(from_, to, func_name):
    """Creates a method(i.e., bound func) to be set on 'from_', which activates 'func_name'
    on 'to'.
    """
    # noinspection PyUnusedLocal
    def forwarded_method(self_, *args, **kwargs):
        return getattr(to, func_name)(*args, **kwargs)

    return types.MethodType(forwarded_method, from_)
```

meta_prog_utils.py

```
def create_proxy_property(property_name, target_name, is_settable=False):
    """
    Creates a property object which forwards "name" to target.
    """
    # noinspection PyUnusedLocal
    def _proxy_get(self):
        return getattr(getattr(self, target_name), property_name)

    # noinspection PyUnusedLocal
    def _proxy_set(self, val):
        return setattr(getattr(self, target_name), property_name, val)

    if not is_settable:
        return property(_proxy_get)
    else:
        return property(_proxy_get, _proxy_set)
```

For properties we use specific logic

webdriver_recorder.py

```python
class RecordingWebElement(object):
    """A wrapper for selenium web element. This enables us to be notified about actions/events for this element."""
    _METHODS_TO_REPLACE = ['find_element', 'find_elements']
    _READONLY_PROPERTIES = ['tag_name', 'text', 'location_once_scrolled_into_view', 'size',
                            'location', 'parent', 'id', 'rect', 'screenshot_as_base64', 'screenshot_as_png']

    def __init__(self, recorder, driver, element):
        self.element = element
        self._recorder = recorder
        self._driver = driver
        # Copies the web element's interface
        create_proxy_interface(self, element, self._READONLY_PROPERTIES)
        # Setting properties
        for attr in self._READONLY_PROPERTIES:
            setattr(self.__class__, attr, create_proxy_property(attr, 'element'))

    def find_element(self, by=By.ID, value=None):
        # Get element from the original implementation of the underlying driver.
        element = self.element['find_element'](by, value)
        # Wrap the element.
        if element:
            element = RecordingWebElement(self._recorder, self._driver, element)
        return element

    def find_elements(self, by=By.ID, value=None):
        # Get result from the original implementation of the underlying driver.
        elements_list = self.element['find_elements'](by, value)
        # Wrap all returned elements.
        if elements_list:
            updated_list = []
            for element in elements_list:
                updated_list.append(RecordingWebElement(self._recorder, self._driver, element))
            elements_list = updated_list
        return elements_list

    def click(self):
        self._recorder.on_click(self)
        self.element.click()
        self._recorder.on_navigate_to_url(self._driver.current_url)

    def send_keys(self, *value):
        text = u''
        for val in value:
            if isinstance(val, int):
                val = val.__str__()
            text += val.encode('utf-8').decode('utf-8')
        self._recorder.on_send_keys(self, text)
        self.element.send_keys(*value)
```

```python
class RecordingWebDriver(object):
    """A wrapper for selenium web driver which creates wrapped elements, and notifies us about events / actions."""
    # Properties require special handling since even testing if they're callable "activates"
    # them, which makes copying them automatically a problem.
    _READONLY_PROPERTIES = ['application_cache', 'current_url', 'current_window_handle',
                            'desired_capabilities', 'log_types', 'name', 'page_source', 'title',
                            'window_handles', 'switch_to', 'mobile', 'application_cache', 'log_types']
    _SETTABLE_PROPERTIES = ['orientation', 'file_detector']

    def __init__(self, recorder, driver):
        self._recorder = recorder
        self.driver = driver
        # Creating the rest of the driver interface by simply forwarding it to the underlying
        # driver.
        metaprog_utils.create_proxy_interface(self, driver,
                                              self._READONLY_PROPERTIES + self._SETTABLE_PROPERTIES)
        for attr in self._READONLY_PROPERTIES:
            if not hasattr(self.__class__, attr):
                setattr(self.__class__, attr, metaprog_utils.create_proxy_property(attr, 'driver'))
        for attr in self._SETTABLE_PROPERTIES:
            if not hasattr(self.__class__, attr):
                setattr(self.__class__, attr, metaprog_utils.create_proxy_property(attr, 'driver', True))

    def get(self, url):
        self._recorder.on_navigate_to_url(url)
        return self.driver.get(url)

    def find_element(self, by=By.ID, value=None):
        # Get element from the original implementation of the underlying driver.
        element = self.driver.find_element(by, value)
        # Wrap the element.
        if element:
            element = RecordingWebElement(self._recorder, self, element)
        return element

    def find_elements(self, by=By.ID, value=None):
        # Get result from the original implementation of the underlying driver.
        elements_list = self.driver.find_elements(by, value)
        # Wrap all returned elements.
        if elements_list:
            updated_results = []
            for element in elements_list:
                updated_results.append(RecordingWebElement(self._recorder, self, element))
            elements_list = updated_results
        return elements_list

    def find_element_by_id(self, id_):
        return self.find_element(by=By.ID, value=id_)

    def find_elements_by_id(self, id_):
```

```python
# noinspection PyAttributeOutsideInit
class Recorder(object):
    """A class for handling the recording"""

    def start(self, driver):
        self._pages = []
        self._current_page = {}
        return RecordingWebDriver(self, driver)

    def close(self):
        self._pages.append(self._current_page)

    def on_navigate_to_url(self, url):
        if self._current_page:
            self._pages.append(self._current_page)

        self._current_page = {'url': url, 'recorded_events': []}

    @staticmethod
    def _get_event_location(element):
        element_location = element.location
        element_size = element.size
        return {'x': element_location['x'] + (element_size['width'] / 2),
                'y': element_location['y'] + (element_size['height'] / 2)}

    def _add_event(self, element, event):
        event['location'] = Recorder._get_event_location(element)
        self._current_page['recorded_events'].append(event)

    def on_click(self, element):
        self._add_event(element, {'event_type': 'click'})

    def on_send_keys(self, element, text):
        self._add_event(element, {'event_type': 'send_keys', 'text': text})

    def export(self):
        return json.dumps(self._pages)
```

Video: our recorder in action

back to selenium/webdriver/remote/webdriver.py

(told you it would be a short break)

selenium/webdriver/remote/webdriver.py

- Initialization code
- A bunch of commands (get/set cookies, get screenshot etc.)
- Navigation methods
- A lot of “find_elementXXXX” variants
- A bunch of properties

Ahh, we already dealt with that :)

1

2

3

Real use case video

Thanks! Hope it was interesting :-O

Github for the code example:
https://github.com/danielputerman/PyCon2016-HTSW